University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 2: this Friday at 11:59 pm
  - Lab 1: Tuesday 1/31 at 11:59 pm
- Assignment 1 has been posted
  - Due on 2/17 at 11:59 pm on GradeScope
  - TAs will talk about the assignment in this week's recitations
  - Video explanation will come out soon

Previous lecture

- Backtracking solution of the Boggle game
- Backtracking framework

Today …

- How to make the non-recursive work constant
  - Backtracking solution of Boggle
- Searching Problem
- Tree ADT

Non-recursive Work

```
void solve(row and column of current cell, word string so far) {
   for each of the eight directions {
      if neighbor down the direction is a in the board and hasn't been used {
         append neighbor's letter to word string and mark neighbor used
         if word string a word with 3+ letters
            add word string to set of solutions
         if word string is a prefix
            solve(row and column of neighbor, word string)
          delete last letter of word string and mark neighbor as unused
      }
   }
}
```

How to make the non-recursive work constant?

- How can we make the dictionary lookup (for prefixes and full words) *O(1)?*
  - Hash table? runtime?
  - Later in the course, we will see an efficient way to perform this task using a *tree*
- How about the time to append and delete letters from the word string?

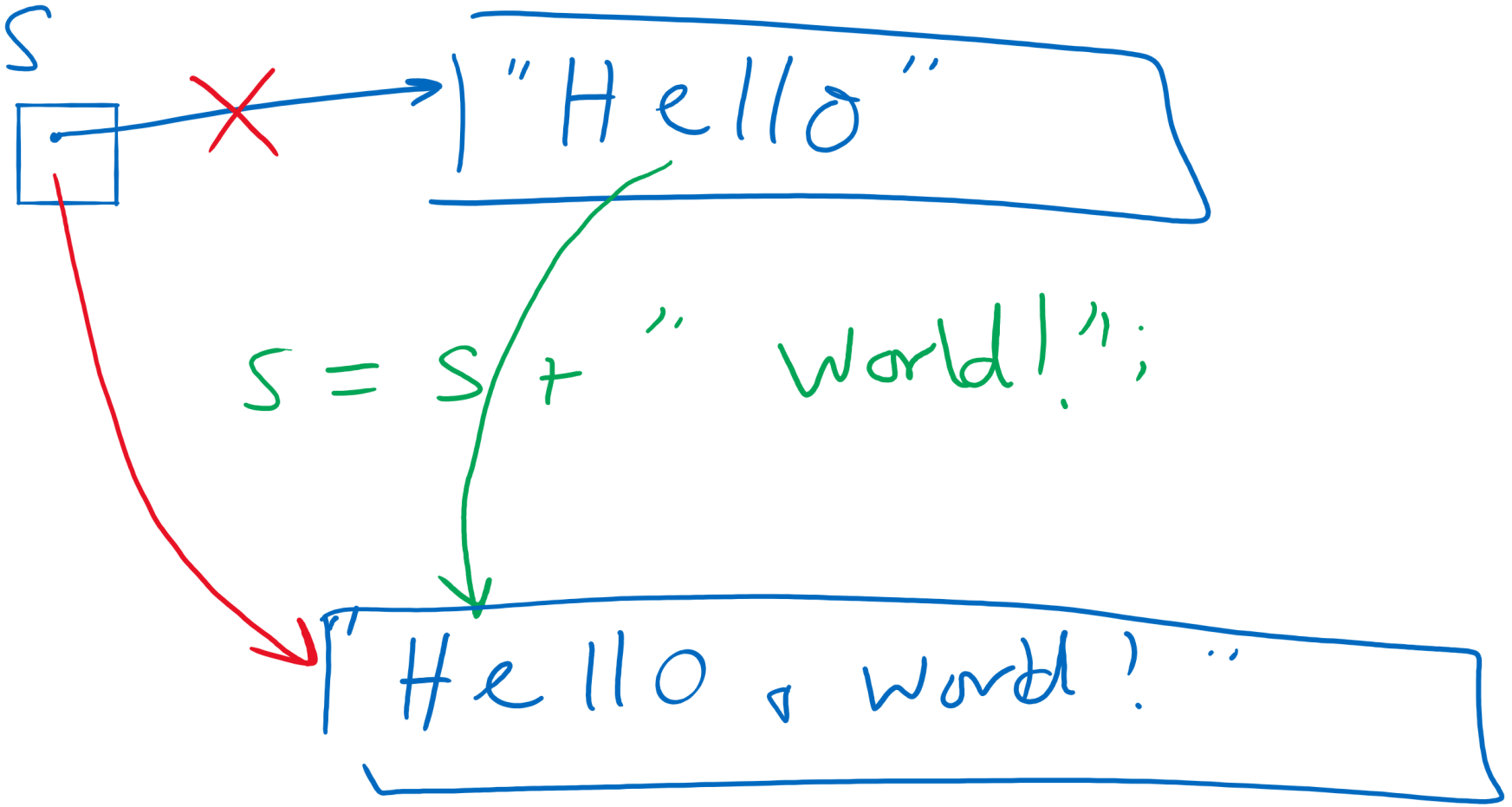
How to make the non-recursive work constant?

- Constructing the words over the course of recursion will mean building up and tearing down strings
  - Moving down the tree adds a new character to the current word string
  - Backtracking removes the most recent character
  - Basically pushing/popping to/from a string stack
- Push/Pop stack operations are generally $\Theta(1)$
  - Unless you need to resize, but that cost can be **amortized**

How to make the non-recursive work constant?

- What if we use String to hold the current word string?
- Java Strings are *immutable*
  - s = new String(“Here is a basic string”);
  - s = s + “ this operation allocates and initializes all over again”;
  - Becomes essentially a Θ(n) operation
    - Where n is the length() of the string

Concatenating to String Objects



StringBuilder to the rescue

- For StringBuilder objects
  - append() and deleteCharAt() can be used to push and pop
  - Back to Θ(1)!
    - Still need to account for resizing, though…
- StringBuffer can also be used for this purpose
  - Differences?

Searching Problem

- Input:
  - a (large) dynamic set of data items in the form of
    - (key, value) pairs
  - a target *key* to search for
  - The input size (*n*) is the number of pairs
    - Key size is assumed to be constant
    - Was that case for Boggle?
- Output:
  - if *key* exists in the set: return the corresponding value
  - otherwise, return key *not found*
- What does dynamic mean?
- How would you implement “key not found”?

Let's create an Abstract Data Type!

- The Symbol Table ADT
  - A set of (key, value) pairs
  - Name dates to earlier use in Compilers
- Operations of the ST ADT
  - insert
  - search
  - delete

Symbol Table Implementations

Implementation	Runtime for Insert	Runtime for search	Runtime for delete
Unsorted Array			
Sorted Array			
Unsorted Linked List			
Sorted Linked List			
Hash Table			

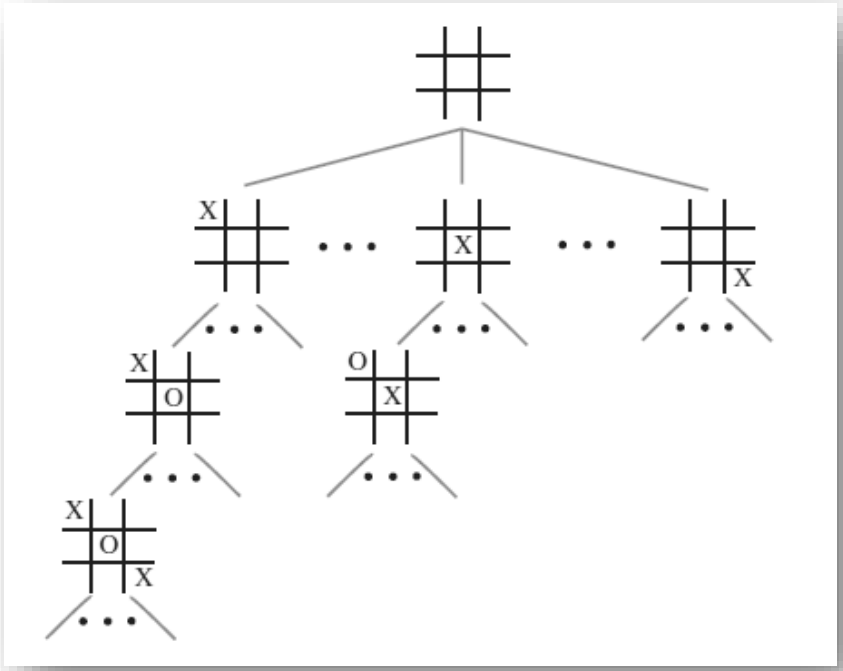
Symbol Table Implementations

Implementation	Runtime for Insert	Runtime for search	Runtime for delete
Unsorted Array	*O(1)* *amortized*	*O(n)*	*O(n)* *amortized*
Sorted Array	*O(n)*	*O(log n)*	*O(n)*
Unsorted Linked List	*O(1)*	*O(n)*	*O(n)*
Sorted Linked List	*O(n)*	*O(n)*	*O(n)*
Hash Table	*O(1) avg.*	*O(1) avg.*	*O(1) avg.*

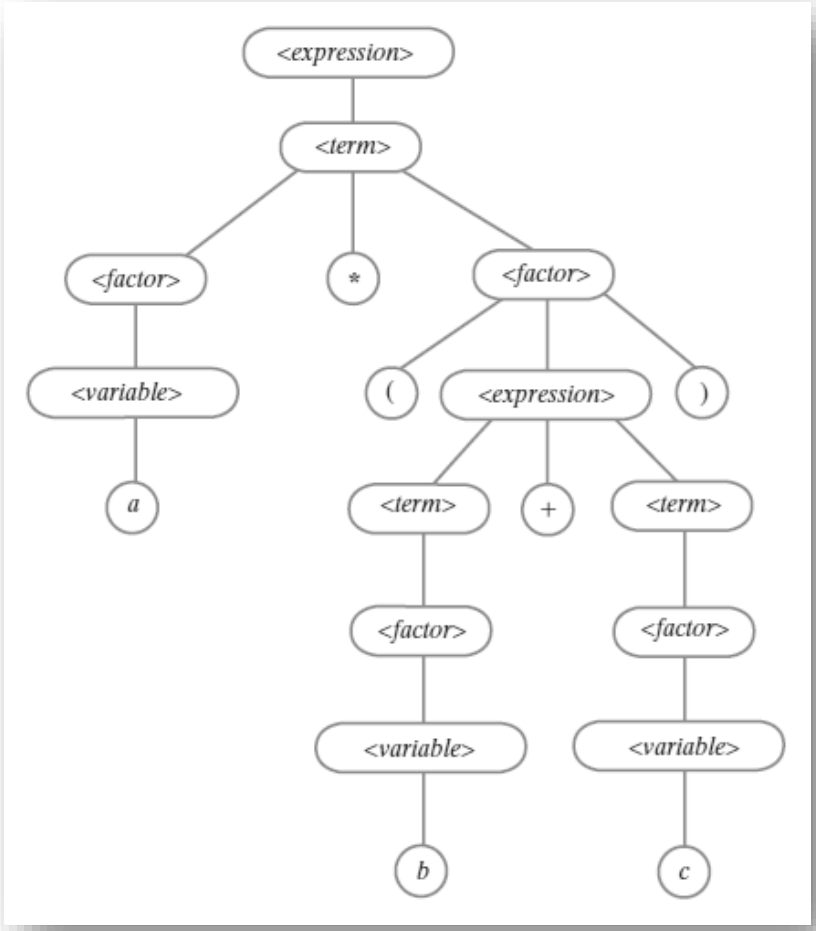
Symbol Table Implementations

- Arrays and Linked Lists are linear structures
- What if we use a non-linear data structure?
  - a Tree?

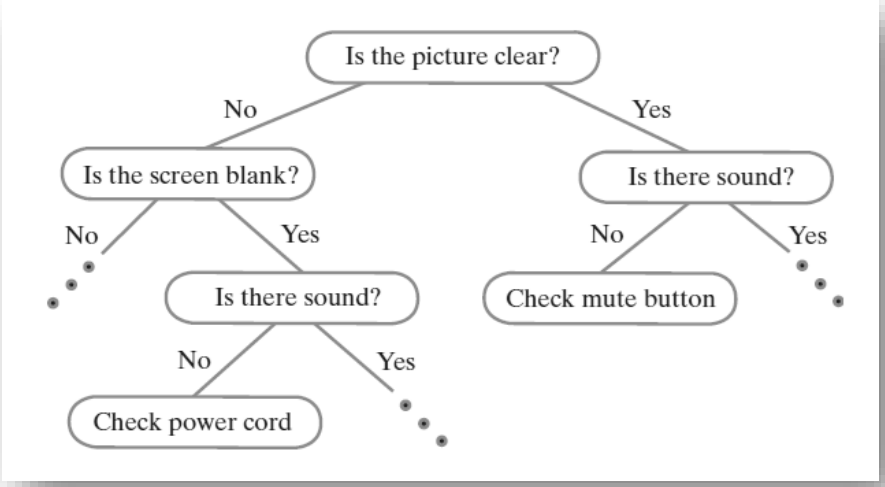
Examples of Trees



Game Tree

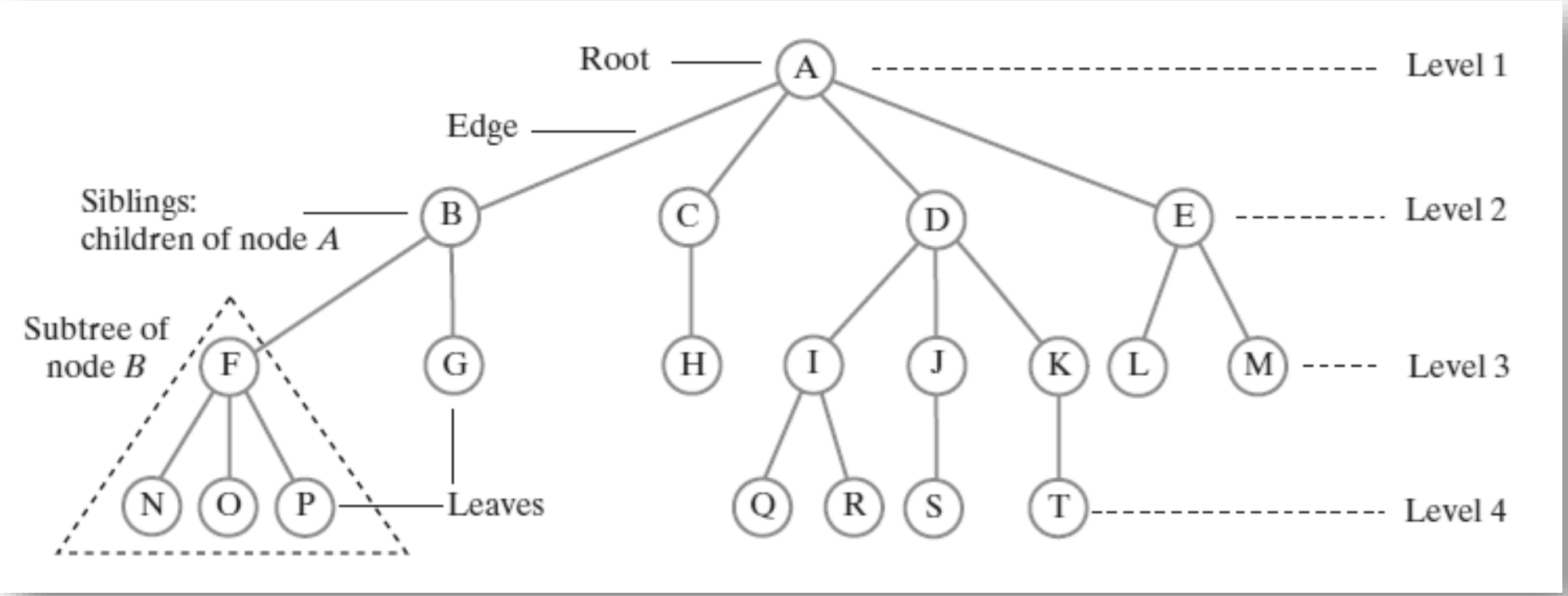


Parse Tree



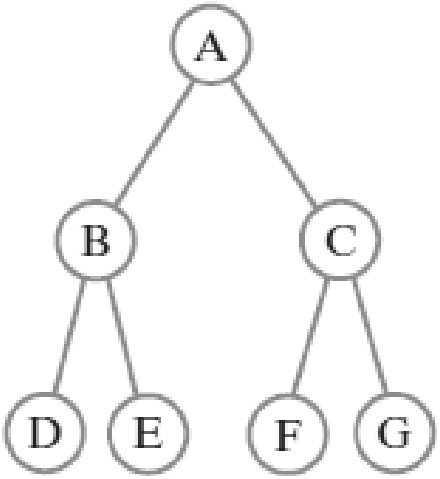
Decision Tree

Tree Terminology



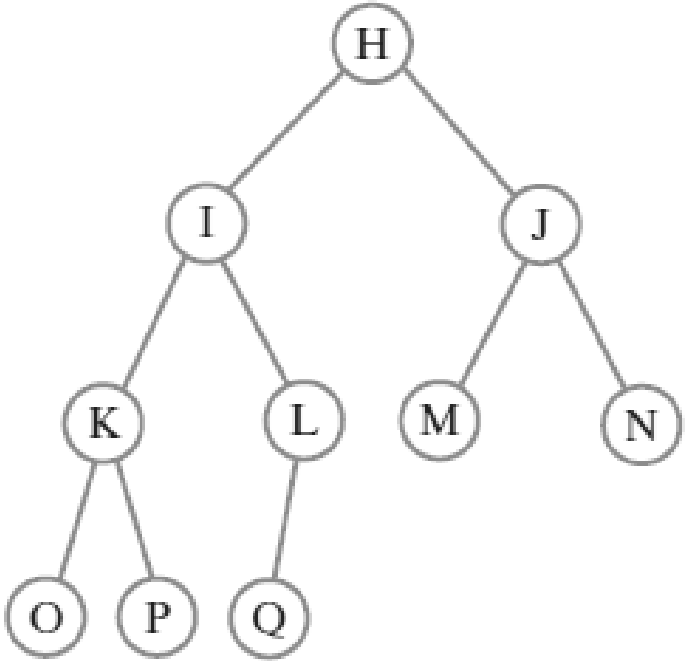
Binary Trees

(a) Full tree

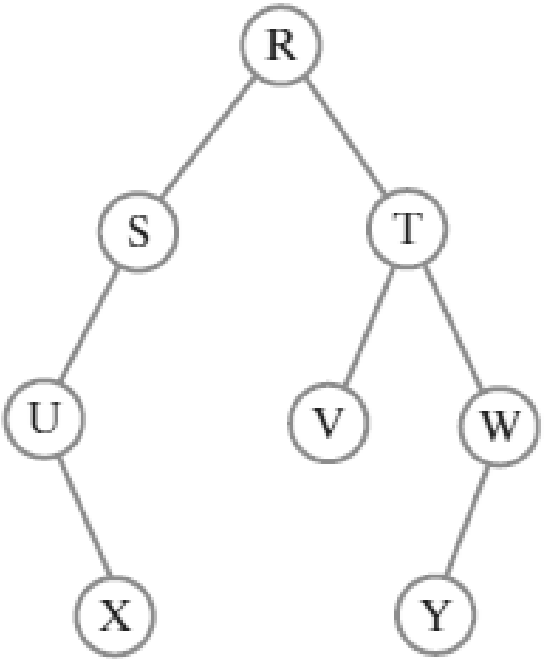


Left children: B, D, F
Right children: C, E, G

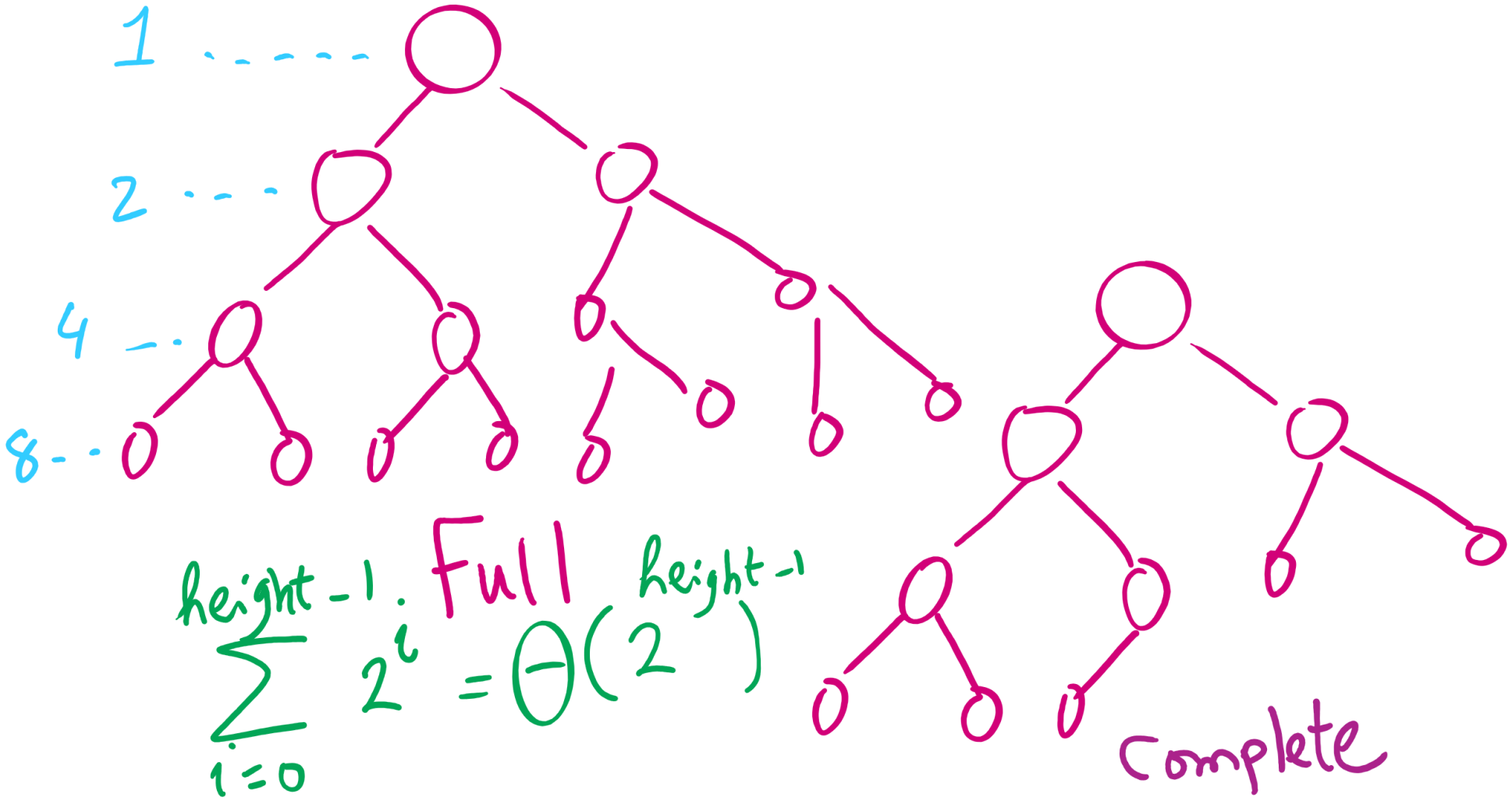
(b) Complete tree



(c) Tree that is not full and not complete



Full vs. Complete Tree



Tree Implementation: Code Walkthrough

- Available online at:
  - https://cs1501-2231.github.io/slides-handouts/CodeHandouts/TreeADT/Slides
  - The slides are under the CodeHandouts/TreeADT/slides folder in the handout repository
  - https://github.com/cs1501-2234/slides-handouts

BinaryNode